An introduction into Verilog simulation

Christian Krieg

October 9, 2024

Welcome to this year's *Digital Integrated Circuits* course! Throughout this course, we will have to solve some assignments that will teach us fundamental concepts of integrated circuit design.

We will demonstrate digital integrated circuit design on a step-by-step simple example of a digital counter. As the name of this course suggests, our ultimate goal is to implement this counter as a digital integrated circuit. In the lab, we map this counter to a field-programmable gate array (FPGA) architecture, and run it on the board shown in Figure 1. The counter's functionality is executed on the FPGA. We will be able to control execution by pushing the buttons, and to visualize the counter value using the light-emitting diodes (LEDs). Because operating a hardware counter with buttons and LEDs may be inconvenient, we also will interface the hardware counter over a universal asynchronous receiver/transmitter (UART), so that we may be able to further process the counter value on a host computer. But this will all happen during the lab week at the end of the exercises.

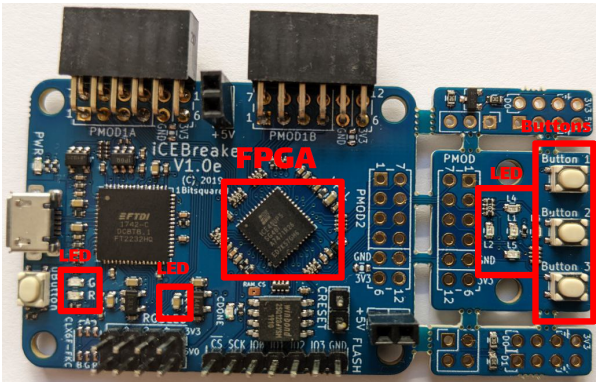


Figure 1: The *icebreaker* board, which will run our counter

Before we can bring our counter on hardware, it is important that we learn how to design digital circuits using state-of-the-art (and beyond) methods and tools, and to get a feeling for the results of our design decisions. We will draw attention on the results of the synthesis steps, which we will verify with several tools along the digital design flow.

Generally speaking, we perform the following steps:

1. **Specify the counter's behavior**
2. Model (implement) the counter in a hardware description language (HDL)
3. Verify the counter's implementation
4. Synthesize and optimize the counter's HDL model + verify
5. Map the counter to the target (FPGA) technology + verify
6. Generate an FPGA bitstream + verify
7. Configure the FPGA with the bitstream + verify
8. Run the counter on hardware

So, let's get started! Our first task is to specify a synchronous counter that runs in two modes, based on a *mode* switch. The two modes are called *up* and *down*. In *up* mode, the counter increments the counter value by **5** at the rising edge of the clock signal, and in *down* mode, it decrements the counter value by **9** at the rising edge of the clock signal. The counter should implement synchronous reset behavior, such that it is set to an initial value of **-50** upon reset. The counter should never exceed a value of **235**, and should never go below a value of **-230** (the counter value sticks near the

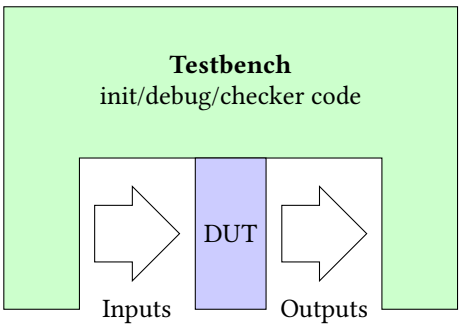


Figure 2: Basic concept of *testbench* and DUT

minimum/maximum value, until the counting direction changes). The counter value should never be **-11** (the counter should jump over this value by one increment/decrement).

Testbenches: A quick tutorial

Before we can bring our counter on hardware, we first have to specify its behavior; then, we have to model it using an HDL; and finally, we have to verify that the counter correctly models its specification. There are various methods available to verify the behavior of HDL models. Today, we will learn about a widely used, very popular verification method: *simulation*. In simulation, a *simulator* takes as input an HDL model of a digital design, along with code that tests the functional correctness of this model. Such testing code is commonly referred to as a *testbench*, and is typically implemented in a separate file to maintain modularity. A testbench tests whether the HDL model, which is commonly known as design under test (DUT), or unit under test (UUT), conforms to its specification. A testbench does not have inputs or outputs. Its only purpose is to provide inputs to a DUT, and to check the DUT's outputs against expected values. Figure 2 shows a typical visualization of a testbench that instantiates and initializes a DUT, provides inputs to the DUT, reads outputs from the DUT, and checks whether the output values conform to the specification. Ideally, a testbench may provide log messages to help debugging the design.

The two major HDLs, VHDL and Verilog, both support simulation. These languages provide constructs to generate input valuations, and functionality to interact with a user and/or simulator (e.g., printing on a screen, reading a file with input data, etc.). The main purpose of an HDL is to model hardware, which means that the final output is a netlist. Although simulation features of HDLs may be very practical to examine a design, they cannot be represented as a netlist—they are not *synthesizable*. For instance, there is no netlist representation for routines like "*read a file from the file system*", or "*print a message to the screen*". HDL language subsets that may be used to model synthesizable hardware are called *synthesizable subsets*. A DUT may only contain synthesizable code. Simulation code may only be used in a testbench.

In the following, we develop a testbench step-by-step that may be used for verification. We use Verilog to implement this testbench.

Design specification

Before we start developing a testbench, we first need to define the reference we are testing against: the design's *specification*. We take the informal, human-language description of the counter from above, and formulate properties that constrain the behavior of our counter. We name the counter's input/output signals as follows:

Table 1: Signal names for the counter

Signal name	Description	Width
cnt	Counter value	10
mode	Counter mode	1
clk	Clock	1
rst	Reset	1

If the *mode* signal is '0', the counter is in *down* mode, and in *up* mode otherwise. We specify the counter's behavioral properties as follows:

1. The counter value always is in the specified range $[-230, 235]$, and it never takes the *invalid* value -11

2. The counter increment is always 5, except when the *invalid* value is jumped over (then, it's twice as big)
3. The counter decrement is always 9, except when the *invalid* value is jumped over (then, it's twice as big)
4. The increment/decrement can be different after the first clock cycle, when *rst* goes from '1' to '0'
5. The increment/decrement can be different when the counter value remains near the minimum/maximum value (i.e., the increment/decrement is 0)

Our challenge is now to develop a testbench which verifies that the counter behaves as specified above.

The following examples assume parameters as given in Table 3.

Table 2: Parameters for the example counter

Parameter	Value
MIN	-287
MAX	309
INV	75
INIT	13
INC	4
DEC	10

A very simple testbench

By convention, each Verilog *module* (e.g., *counter*, like the DUT we are testing) is implemented in its own Verilog file (e.g., *counter.v*), and each Verilog module is tested by its own testbench, which gets the same filename like its DUT, along with a prefix or suffix that identifies it as a testbench, for instance "*tb_*", "*test_*", etc. So, if our counter is implemented in a Verilog module *counter*, written into a file *counter.v*, its associated testbench module is named *tb_counter*, written into a file *tb_counter.v*.

Listing 1 shows a very simple testbench written in Verilog. Line 1 gives a physical meaning to simulation time, and tells the simulator how to interpret time. The first value specifies the time unit, and the second value specifies the granularity of time, which is important for the simulator to correctly round time values. The testbench itself is implemented in a Verilog module (Lines 3 to 42), without input or output ports that connect to the outside world. While not having own inputs or outputs, the testbench defines registers to store input values provided to the DUT in Lines 6 to 8, and a wire that carries the DUT's output value in Line 11. Lines 14 to 19 show the DUT's instantiation, along with a port map that connect the input/output signals of the DUT with the previously defined input/output signals of the testbench. Line 22 shows an *always* block that generates values for the clock signal by toggling its value every 5 time steps. Lines 25 to 38 show an *initial* block that assigns values to input signals after different amounts of time steps, given after the "hash" operator "#". This way, a reset sequence is implemented in Lines 28 to 29, and the *mode* input is alternated in Lines 32 to 35 to change the DUT's counting direction (up/down) The *$finish* statement in Line 36 tells the simulator to end the simulation at this point. Finally, Line 40 implements a directive to print the counter value (which is the DUT's output value) every time it changes.

Listing 1: A very simple Verilog testbench *tb_counter-1.v*

```
1  `timescale 1ns/1ns
2
3  module tbcounter;
4
5     // Define inputs
6     reg clk=0;
7     reg rst=0;
8     reg mode=0;
9
10    // Define outputs
11    wire signed [9:0] cnt;
12
13    // Instantiate design under test (DUT)
14    counter dut (
15       .clk(clk),
16       .rst(rst),
17       .mode(mode),
18       .cnt(cnt)
19    );
20
21    // Generate clock
22    always #5 clk = !clk;
23
24    // Stimulate the DUT (vary input values)
25    initial begin
26
```

```
27      // Reset the DUT
28      #5 rst = 1;
29      #10 rst = 0;
30
31      // Operate the DUT
32      #17 mode = 1;
33      #22 mode = 0;
34      #12 mode = 1;
35      #100 mode = 0;
36      $finish;
37
38    end
39
40    initial $monitor("At time %0t, value = %h (%0d)", $time, cnt, cnt);
41
42 endmodule
```

We simulate the DUT along with the testbench given in Listing 1. There are several Verilog simulators available, both commercial and open-source. For the following examples, we use the free and open-source Verilog simulator *Icarus Verilog*. Listing 2 invokes *Icarus Verilog*, and provides log output from the simulation.

Running the simulation happens in two steps: First, we call the *Icarus Verilog* compiler *iverilog* in Line 1, which translates the HDL model into an executable program, with the following options and arguments:

Table 3: Options/Arguments to *iverilog*

Option/argument	**Description**
`-l ‘yosys-config --datdir/simcells.v‘`	Provide *Yosys* cell library
`-o counter`	Define the target name
`counter.v`	The name of the file that implements the DUT
`tbcounter-1.v`	The name of the file that implements the testbench

When *iverilog* successfully compiles the executable program, we call the *Icarus Verilog vvp runtime engine* to execute the simulation in Line 2. The simulator's debug output is shown in Lines 3 to 19, providing counter values both in hexadecimal and decimal form. We notice the counter counting up and down, recognize the initial value (13), the increment (4), and the decrement (10). In Line 20, the simulator reports the end of simulation.

Listing 2: Simulating the testbench given in Listing 1

```
1  $ iverilog -l ‘yosys-config --datdir/simcells.v‘ -o counter counter.v tbcounter-1.v
2  $ vvp counter
3  At time 0, value = 00d (13)
4  At time 15, value = 003 (3)
5  At time 25, value = 3f9 (-7)
6  At time 35, value = 3fd (-3)
7  At time 45, value = 001 (1)
8  At time 55, value = 3f7 (-9)
9  At time 65, value = 3ed (-19)
10 At time 75, value = 3f1 (-15)
11 At time 85, value = 3f5 (-11)
12 At time 95, value = 3f9 (-7)
13 At time 105, value = 3fd (-3)
14 At time 115, value = 001 (1)
15 At time 125, value = 005 (5)
16 At time 135, value = 009 (9)
17 At time 145, value = 00d (13)
18 At time 155, value = 011 (17)
19 At time 165, value = 015 (21)
20 tbcounter-1.v:36: $finish called at 166 (1ns)
```

While we get an idea what happens inside our counter, we are fairly limited with this simple testbench:

1. The testbench only reports, but does not verify
2. The coverage may be very limited
3. We see the simulation results printed to standard output, but it is nowhere recorded for later use

In the following, we gradually extend our testbench to address our requirements.

Recording simulation results

As noted before, we do not have any opportunity to analyze simulation results at a later time. This is why we add another *initial* block to our testbench in Lines 6 to 9, which invokes routines to dump simulation data.

Listing 3: Adding support for dumping simulation data (*tb_counter-2.v*)

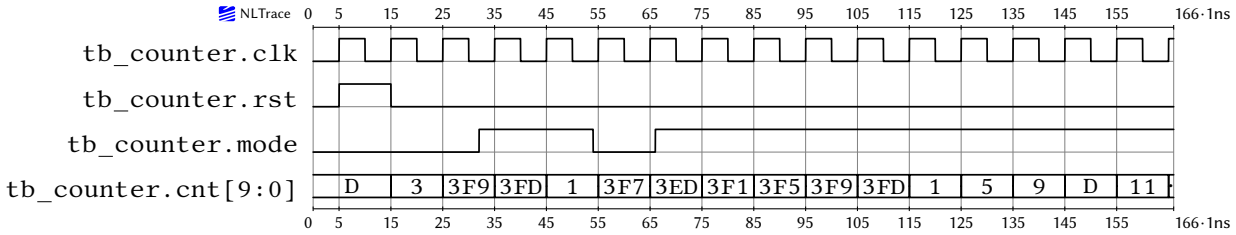
```
1  `timescale 1ns/1ns
2
3  module tbcounter;
4
5     // Specify dump file and the scope to be dumped
6     initial begin
7        $dumpfile("counter.vcd");
8        $dumpvars(0, tbcounter);
9     end
10
11    // Define inputs
12    reg clk=0;
13    reg rst=0;
14    reg mode=0;
15
16    // Define outputs
17    wire signed [9:0] cnt;
18
19    // Instantiate design under test (DUT)
20    counter dut (
21       .clk(clk),
22       .rst(rst),
23       .mode(mode),
24       .cnt(cnt)
25    );
26
27    // Generate clock
28    always #5 clk = !clk;
29
30    // Stimulate the design (vary input values)
31    initial begin
32
33       // Reset the DUT
34       #5 rst = 1;
35       #10 rst = 0;
36
37       // Operate the DUT
38       #17 mode = 1;
39       #22 mode = 0;
40       #12 mode = 1;
41       #100 mode = 0;
42       $finish;
43
44    end
45
46    initial $monitor("At time %0t, value = %h (%0d)", $time, cnt, cnt);
47
48 endmodule
```

When we run the simulation again, the simulator reports about opening a file for dumping simulation results (Line 3 in Listing 4).

Listing 4: Simulating the testbench given in Listing 3

```
1  $ iverilog -l `yosys-config --datdir/simcells.v` -o counter counter.v tbcounter-2.v
2  $ vvp counter
3  VCD info: dumpfile counter.vcd opened for output.
4  At time 0, value = 00d (13)
5  At time 15, value = 003 (3)
6  At time 25, value = 3f9 (-7)
7  At time 35, value = 3fd (-3)
8  At time 45, value = 001 (1)
9  At time 55, value = 3f7 (-9)
10 At time 65, value = 3ed (-19)
11 At time 75, value = 3f1 (-15)
12 At time 85, value = 3f5 (-11)
13 At time 95, value = 3f9 (-7)
14 At time 105, value = 3fd (-3)
15 At time 115, value = 001 (1)
16 At time 125, value = 005 (5)
17 At time 135, value = 009 (9)
18 At time 145, value = 00d (13)
19 At time 155, value = 011 (17)
20 At time 165, value = 015 (21)
21 tbcounter-2.v:36: $finish called at 166 (1ns)
```

With that file, we are now able to visualize simulation results using a waveform diagram:



Also, we can interactively inspect the waveforms in a waveform analyzer, such as *GTKWave* (Figure 3):

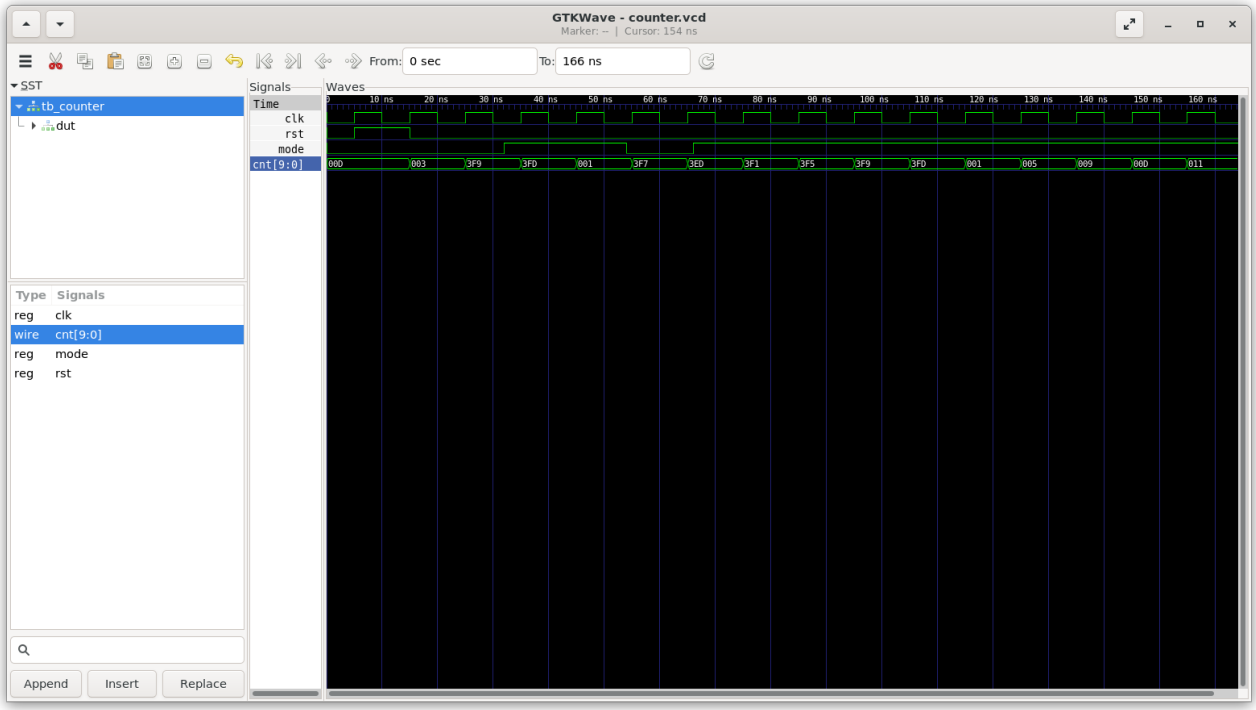


Figure 3: Interactive waveform analysis with *GTKWave*

By just adding four lines of code to our testbench, we are much more powerful in using the simulation results.

Checking values

Let us now add another line of code to our testbench, which enables us to check the value of the counter. In Lines 46 and 49 in Listing 5 we add two *assert* statements, which test the counter value at the actual time step. From Line 20 in Listing 3 we know that the counter value in the last time step is `0x15`. In the first *assert* statement in Line 46 in Listing 5, we check the counter value for the correct value, which silently passes in Listing 6. In the second *assert* statement in Line 49 in Listing 5, we deliberately check against an incorrect value, which errors out in Lines 21 and 22 in Listing 6.

Listing 5: Checking values of DUT output (*tb_counter-3.v*)

```
1  `timescale 1ns/1ns
2
3  module tbcounter;
4
5     // Specify dump file and the scope to be dumped
6     initial begin
7        $dumpfile("counter.vcd");
8        $dumpvars(0, tbcounter);
9     end
10
11    // Define inputs
12    reg clk=0;
13    reg rst=0;
14    reg mode=0;
15
16    // Define outputs
17    wire signed [9:0] cnt;
18
19    // Instantiate design under test (DUT)
```

```
20    counter dut (
21       .clk(clk),
22       .rst(rst),
23       .mode(mode),
24       .cnt(cnt)
25    );
26
27    // Generate clock
28    always #5 clk = !clk;
29
30    // Stimulate the design (vary input values)
31    initial begin
32
33       // Reset the DUT
34       #5 rst = 1;
35       #10 rst = 0;
36
37       // Operate the DUT
38       #17 mode = 1;
39       #22 mode = 0;
40       #12 mode = 1;
41       #100 mode = 0;
42
43       // Verify the counter
44
45       // Bare inline assertion with passing test case
46       assert(cnt == 10'h015);
47
48       // Bare inline assertion with failing test case
49       assert(cnt == 10'h014);
50
51       $finish;
52
53    end
54
55    initial $monitor("At time %0t, value = %h (%0d)", $time, cnt, cnt);
56
57 endmodule
```

When running the simulation, we have to call *iverilog* with support for the *assert* statement. We add the `-g2012` option to its command line, which selects IEEE1800-2012 language support.

Listing 6: Simulating the testbench given in Listing 5

```
1  $ iverilog -g2012 -l `yosys-config --datdir/simcells.v` -o counter counter.v tbcounter-3.v
2  $ vvp counter
3  VCD info: dumpfile counter.vcd opened for output.
4  At time 0, value = 00d (13)
5  At time 15, value = 003 (3)
6  At time 25, value = 3f9 (-7)
7  At time 35, value = 3fd (-3)
8  At time 45, value = 001 (1)
9  At time 55, value = 3f7 (-9)
10 At time 65, value = 3ed (-19)
11 At time 75, value = 3f1 (-15)
12 At time 85, value = 3f5 (-11)
13 At time 95, value = 3f9 (-7)
14 At time 105, value = 3fd (-3)
15 At time 115, value = 001 (1)
16 At time 125, value = 005 (5)
17 At time 135, value = 009 (9)
18 At time 145, value = 00d (13)
19 At time 155, value = 011 (17)
20 At time 165, value = 015 (21)
21 ERROR: tbcounter-3.v:49:
22      Time: 166 Scope: tbcounter
23 tbcounter-3.v:51: $finish called at 166 (1ns)
```

The error message given in Line 21 in Listing 6 leads us to the failing assertion in the file *tb_counter-3.v*, Line 49, which tells us that the counter value is not `0x14` at the respective time step. This is valuable debug information.

Checking first properties of our counter

We now have the tools available to check some properties of our counter, which must hold at every time step (or, *at every clock cycle*). We therefore add an *always* block in Lines 50 to 54 to our testbench given in Listing 7, holding assertions that check some of the counter's properties. The assertion in Line 51 checks at every clock cycle the counter value being not smaller than the specified minimum value; the assertion in Line 52 checks the counter value being not larger than the specified maximum value; the assertion in Line 53 checks the counter value being not equal to the specified invalid value.

Listing 7: Checking first properties of our counter (*tb_counter-4.v*)

```
1  `timescale 1ns/1ns
```

```
2
3  module tbcounter;
4
5     // Specify dump file and the scope to be dumped
6     initial begin
7        $dumpfile("counter.vcd");
8        $dumpvars(0, tbcounter);
9     end
10
11    // Define inputs
12    reg clk=0;
13    reg rst=0;
14    reg mode=0;
15
16    // Define outputs
17    wire signed [9:0] cnt;
18
19    // Instantiate design under test (DUT)
20    counter dut (
21       .clk(clk),
22       .rst(rst),
23       .mode(mode),
24       .cnt(cnt)
25    );
26
27    // Generate clock
28    always #5 clk = !clk;
29
30    // Stimulate the design (vary input values)
31    initial begin
32
33       // Reset the DUT
34       #5 rst = 1;
35       #10 rst = 0;
36
37       // Operate the DUT
38       #17 mode = 1;
39       #22 mode = 0;
40       #12 mode = 1;
41       #100 mode = 0;
42
43       $finish;
44
45    end
46
47    initial $monitor("At time %0t, value = %h (%0d)", $time, cnt, cnt);
48
49    // Verify properties of the counter
50    always @(posedge clk) begin
51       assert (cnt >= -287);
52       assert (cnt <= 309);
53       assert (cnt != 75);
54    end
55
56 endmodule
```

When running the simulation as shown in Listing 8, we notice that the DUT, our counter, does not violate any of the properties we check. However, looking at the counter values, we notice that the counter value does not even go near any of the values checked for in the assertions, which leaves the expressiveness of the simulation results in a questionable state.

Listing 8: Simulating the testbench given in Listing 7

```
1  $ iverilog -g2012 -l `yosys-config --datdir/simcells.v` -o counter counter.v tbcounter-4.v
2  $ vvp counter
3  VCD info: dumpfile counter.vcd opened for output.
4  At time 0, value = 00d (13)
5  At time 15, value = 003 (3)
6  At time 25, value = 3f9 (-7)
7  At time 35, value = 3fd (-3)
8  At time 45, value = 001 (1)
9  At time 55, value = 3f7 (-9)
10 At time 65, value = 3ed (-19)
11 At time 75, value = 3f1 (-15)
12 At time 85, value = 3f5 (-11)
13 At time 95, value = 3f9 (-7)
14 At time 105, value = 3fd (-3)
15 At time 115, value = 001 (1)
16 At time 125, value = 005 (5)
17 At time 135, value = 009 (9)
18 At time 145, value = 00d (13)
19 At time 155, value = 011 (17)
20 At time 165, value = 015 (21)
21 tbcounter-4.v:43: $finish called at 166 (1ns)
```

Adding test cases

As we noticed earlier, the expressiveness of a simulation strongly depends on the values we check. In order to increase the expressiveness of simulation results generated by our testbench, the counter value must become close to the minimum, the maximum, and the invalid value. The only input values we have control over is the simulation time and the *mode* signal. We therefore increase simulation time in Listing 9, Lines 39 and 40, and vary the *mode* signal such that the counter value travels to the minimum and maximum values. In order to keep verification time low, we first count up, because the counter's increment is nearly half the value of its decrement, so we like to start counting up from near the reset value rather than counting up from the minimum value.

Listing 9: Increasing the number of test cases (*tb_counter-5.v*)

```
1  `timescale 1ns/1ns
2
3  module tbcounter;
4
5     // Specify dump file and the scope to be dumped
6     initial begin
7        $dumpfile("counter.vcd");
8        $dumpvars(0, tbcounter);
9     end
10
11    // Define inputs
12    reg clk=0;
13    reg rst=0;
14    reg mode=0;
15
16    // Define outputs
17    wire signed [9:0] cnt;
18
19    // Instantiate design under test (DUT)
20    counter dut (
21       .clk(clk),
22       .rst(rst),
23       .mode(mode),
24       .cnt(cnt)
25    );
26
27    // Generate clock
28    always #5 clk = !clk;
29
30    // Stimulate the design (vary input values)
31    initial begin
32
33       // Reset the DUT
34       #5 rst = 1;
35       #10 rst = 0;
36
37       // Operate the DUT
38       #17 mode = 1;
39       #800 mode = 0;
40       #800 mode = 1;
41       #100 mode = 0;
42
43       $finish;
44
45    end
46
47    initial $monitor("At time %0t, value = %h (%0d)", $time, cnt, cnt);
48
49    // Verify properties of the counter
50    always @(posedge clk) begin
51       assert (cnt ¿= -287);
52       assert (cnt ¡= 309);
53       assert (cnt != 75);
54    end
55
56 endmodule
```

When simulating the testbench given in Listing 9, we learn from the simulation output given in Listing 10, that the counter reaches its maximum value at time 815 (Line 85), and that it comes near its minimum value at time 1415 (Line 144). In Lines 26, 27, 108 and 109 the counter value is close to its invalid value, but they do not match. We assume that it may be challenging to manually find a sequence of *mode* changes such that the counter value would match its invalid value, in order to verify the counter's correct behavior for that case.

Listing 10: Simulating the testbench given in Listing 9

```
1 $ iverilog -g2012 -l `yosys-config --datdir/simcells.v` -o counter counter.v tbcounter-5.v
2 $ vvp counter
3 VCD info: dumpfile counter.vcd opened for output.
4 At time 0, value = 00d (13)
5 At time 15, value = 003 (3)
6 At time 25, value = 3f9 (-7)
7 At time 35, value = 3fd (-3)
```

```
At time 45, value = 001 (1)
At time 55, value = 005 (5)
At time 65, value = 009 (9)
At time 75, value = 00d (13)
At time 85, value = 011 (17)
At time 95, value = 015 (21)
At time 105, value = 019 (25)
At time 115, value = 01d (29)
At time 125, value = 021 (33)
At time 135, value = 025 (37)
At time 145, value = 029 (41)
At time 155, value = 02d (45)
At time 165, value = 031 (49)
At time 175, value = 035 (53)
At time 185, value = 039 (57)
At time 195, value = 03d (61)
At time 205, value = 041 (65)
At time 215, value = 045 (69)
At time 225, value = 049 (73)
At time 235, value = 04d (77)
At time 245, value = 051 (81)
At time 255, value = 055 (85)
At time 265, value = 059 (89)
At time 275, value = 05d (93)
At time 285, value = 061 (97)
At time 295, value = 065 (101)
At time 305, value = 069 (105)
At time 315, value = 06d (109)
At time 325, value = 071 (113)
At time 335, value = 075 (117)
At time 345, value = 079 (121)
At time 355, value = 07d (125)
At time 365, value = 081 (129)
At time 375, value = 085 (133)
At time 385, value = 089 (137)
At time 395, value = 08d (141)
At time 405, value = 091 (145)
At time 415, value = 095 (149)
At time 425, value = 099 (153)
At time 435, value = 09d (157)
At time 445, value = 0a1 (161)
At time 455, value = 0a5 (165)
At time 465, value = 0a9 (169)
At time 475, value = 0ad (173)
At time 485, value = 0b1 (177)
At time 495, value = 0b5 (181)
At time 505, value = 0b9 (185)
At time 515, value = 0bd (189)
At time 525, value = 0c1 (193)
At time 535, value = 0c5 (197)
At time 545, value = 0c9 (201)
At time 555, value = 0cd (205)
At time 565, value = 0d1 (209)
At time 575, value = 0d5 (213)
At time 585, value = 0d9 (217)
At time 595, value = 0dd (221)
At time 605, value = 0e1 (225)
At time 615, value = 0e5 (229)
At time 625, value = 0e9 (233)
At time 635, value = 0ed (237)
At time 645, value = 0f1 (241)
At time 655, value = 0f5 (245)
At time 665, value = 0f9 (249)
At time 675, value = 0fd (253)
At time 685, value = 101 (257)
At time 695, value = 105 (261)
At time 705, value = 109 (265)
At time 715, value = 10d (269)
At time 725, value = 111 (273)
At time 735, value = 115 (277)
At time 745, value = 119 (281)
At time 755, value = 11d (285)
At time 765, value = 121 (289)
At time 775, value = 125 (293)
At time 785, value = 129 (297)
At time 795, value = 12d (301)
At time 805, value = 131 (305)
At time 815, value = 135 (309)
At time 835, value = 12b (299)
At time 845, value = 121 (289)
At time 855, value = 117 (279)
At time 865, value = 10d (269)
At time 875, value = 103 (259)
At time 885, value = 0f9 (249)
At time 895, value = 0ef (239)
At time 905, value = 0e5 (229)
At time 915, value = 0db (219)
At time 925, value = 0d1 (209)
At time 935, value = 0c7 (199)
At time 945, value = 0bd (189)
At time 955, value = 0b3 (179)
```

```
At time 965, value = 0a9 (169)
At time 975, value = 09f (159)
At time 985, value = 095 (149)
At time 995, value = 08b (139)
At time 1005, value = 081 (129)
At time 1015, value = 077 (119)
At time 1025, value = 06d (109)
At time 1035, value = 063 (99)
At time 1045, value = 059 (89)
At time 1055, value = 04f (79)
At time 1065, value = 045 (69)
At time 1075, value = 03b (59)
At time 1085, value = 031 (49)
At time 1095, value = 027 (39)
At time 1105, value = 01d (29)
At time 1115, value = 013 (19)
At time 1125, value = 009 (9)
At time 1135, value = 3ff (-1)
At time 1145, value = 3f5 (-11)
At time 1155, value = 3eb (-21)
At time 1165, value = 3e1 (-31)
At time 1175, value = 3d7 (-41)
At time 1185, value = 3cd (-51)
At time 1195, value = 3c3 (-61)
At time 1205, value = 3b9 (-71)
At time 1215, value = 3af (-81)
At time 1225, value = 3a5 (-91)
At time 1235, value = 39b (-101)
At time 1245, value = 391 (-111)
At time 1255, value = 387 (-121)
At time 1265, value = 37d (-131)
At time 1275, value = 373 (-141)
At time 1285, value = 369 (-151)
At time 1295, value = 35f (-161)
At time 1305, value = 355 (-171)
At time 1315, value = 34b (-181)
At time 1325, value = 341 (-191)
At time 1335, value = 337 (-201)
At time 1345, value = 32d (-211)
At time 1355, value = 323 (-221)
At time 1365, value = 319 (-231)
At time 1375, value = 30f (-241)
At time 1385, value = 305 (-251)
At time 1395, value = 2fb (-261)
At time 1405, value = 2f1 (-271)
At time 1415, value = 2e7 (-281)
At time 1635, value = 2eb (-277)
At time 1645, value = 2ef (-273)
At time 1655, value = 2f3 (-269)
At time 1665, value = 2f7 (-265)
At time 1675, value = 2fb (-261)
At time 1685, value = 2ff (-257)
At time 1695, value = 303 (-253)
At time 1705, value = 307 (-249)
At time 1715, value = 30b (-245)
At time 1725, value = 30f (-241)
tbcounter-5.v:43: $finish called at 1732 (1ns)
```

Considering the past

By now, we have a considerable testbench which enables us to check some properties of our counter. However, to check the correctness of its increment and decrement, we need to know the counter's value of the previous clock cycle. Also, due to simulation semantics, the counter value only updates at the end of a (simulation) time step in non-blocking statements (which may be assumed in a synchronous design). Therefore, upon reset, the counter is assigned its initial value at the very end of the time step in which the reset occurs, which makes it impossible to test correct reset behavior by comparing the counter value to its initial value when the reset signal is 1. Therfore, we need to observe the reset signal's value of the previous clock cycle. Likewise, we need to know the *mode* signal's value of the previous clock cycle in order to determine from the current counter value if it was correctly incremented or decremented.

The solution is fairly simple: We need to delay signals *rst*, *mode*, and *cnt* by one clock cycle, and store their delayed versions in signals *past_rst*, *past_mode*, and *past_cnt*, as shown in Listing 11, Lines 50 to 57. Having available these past values allows us to check more properties of our counter in Lines 64 to 78.

Listing 11: Checking more properties of our counter, considering past values (*tb_counter-6.v*)

```
1  `timescale 1ns/1ns
2
3  module tbcounter;
4
5     // Specify dump file and the scope to be dumped
6     initial begin
7        $dumpfile("counter.vcd");
8        $dumpvars(0, tbcounter);
```

```
9     end
10
11    // Define inputs
12    reg clk=0;
13    reg rst=0;
14    reg mode=0;
15
16    // Define outputs
17    wire signed [9:0] cnt;
18
19    // Instantiate design under test (DUT)
20    counter dut (
21       .clk(clk),
22       .rst(rst),
23       .mode(mode),
24       .cnt(cnt)
25    );
26
27    // Generate clock
28    always #5 clk = !clk;
29
30    // Stimulate the design (vary input values)
31    initial begin
32
33       // Reset the DUT
34       #5 rst = 1;
35       #10 rst = 0;
36
37       // Operate the DUT
38       #17 mode = 1;
39       #800 mode = 0;
40       #800 mode = 1;
41       #100 mode = 0;
42
43       $finish;
44
45    end
46
47    initial $monitor("At time %0t, value = %h (%0d)", $time, cnt, cnt);
48
49    // Store past values of signals rst, mode, and cnt
50    reg signed [9:0] pastcnt = 'x;
51    reg pastmode = 'x;
52    reg pastrst = 'x;
53    always @(posedge clk) begin
54       pastcnt ¡= cnt;
55       pastrst ¡= rst;
56       pastmode ¡= mode;
57    end
58
59    // Verify properties of the counter
60    always @(posedge clk) begin
61       assert (cnt ¿= -287);
62       assert (cnt ¡= 309);
63       assert (cnt != 75);
64       if(pastrst) begin
65          assert(cnt == 13);
66       end else if (!rst) begin
67          if (pastmode == 1) begin
68
69             // Add the remaining assertions here to check the correct
70             // increment
71
72          end else begin // pastmode == 0
73
74             // Add the remaining assertions here to check the correct
75             // decrement
76
77          end
78       end
79    end
80
81 endmodule
```

Now, let's assume that our counter implements a reset value different than specified, for instance 14 instead of 13. Then, the simulation of Listing 11 results in the output given in Listing 12, with an error message at the very top (Lines 5 and 6), which may be easily overlooked. It would be desirable if the simulation would stop, and the simulator would exit with an exit code different than 0 in case an assertion fails.

Listing 12: Simulating the testbench given in Listing 11

```
1 $ iverilog -g2012 -l `yosys-config --datdir/simcells.v` -o counter counter.v tbcounter-6.v
2 $ vvp counter
3 VCD info: dumpfile counter.vcd opened for output.
4 At time 0, value = 00e (14)
5 ERROR: tbcounter-6.v:72:
6      Time: 15 Scope: tbcounter
7 At time 15, value = 004 (4)
```

```
At time 25, value = 3fa (-6)
At time 35, value = 3fe (-2)
At time 45, value = 002 (2)
At time 55, value = 006 (6)
At time 65, value = 00a (10)
At time 75, value = 00e (14)
At time 85, value = 012 (18)
At time 95, value = 016 (22)
At time 105, value = 01a (26)
At time 115, value = 01e (30)
At time 125, value = 022 (34)
At time 135, value = 026 (38)
At time 145, value = 02a (42)
At time 155, value = 02e (46)
At time 165, value = 032 (50)
At time 175, value = 036 (54)
At time 185, value = 03a (58)
At time 195, value = 03e (62)
At time 205, value = 042 (66)
At time 215, value = 046 (70)
At time 225, value = 04a (74)
At time 235, value = 04e (78)
At time 245, value = 052 (82)
At time 255, value = 056 (86)
At time 265, value = 05a (90)
At time 275, value = 05e (94)
At time 285, value = 062 (98)
At time 295, value = 066 (102)
At time 305, value = 06a (106)
At time 315, value = 06e (110)
At time 325, value = 072 (114)
At time 335, value = 076 (118)
At time 345, value = 07a (122)
At time 355, value = 07e (126)
At time 365, value = 082 (130)
At time 375, value = 086 (134)
At time 385, value = 08a (138)
At time 395, value = 08e (142)
At time 405, value = 092 (146)
At time 415, value = 096 (150)
At time 425, value = 09a (154)
At time 435, value = 09e (158)
At time 445, value = 0a2 (162)
At time 455, value = 0a6 (166)
At time 465, value = 0aa (170)
At time 475, value = 0ae (174)
At time 485, value = 0b2 (178)
At time 495, value = 0b6 (182)
At time 505, value = 0ba (186)
At time 515, value = 0be (190)
At time 525, value = 0c2 (194)
At time 535, value = 0c6 (198)
At time 545, value = 0ca (202)
At time 555, value = 0ce (206)
At time 565, value = 0d2 (210)
At time 575, value = 0d6 (214)
At time 585, value = 0da (218)
At time 595, value = 0de (222)
At time 605, value = 0e2 (226)
At time 615, value = 0e6 (230)
At time 625, value = 0ea (234)
At time 635, value = 0ee (238)
At time 645, value = 0f2 (242)
At time 655, value = 0f6 (246)
At time 665, value = 0fa (250)
At time 675, value = 0fe (254)
At time 685, value = 102 (258)
At time 695, value = 106 (262)
At time 705, value = 10a (266)
At time 715, value = 10e (270)
At time 725, value = 112 (274)
At time 735, value = 116 (278)
At time 745, value = 11a (282)
At time 755, value = 11e (286)
At time 765, value = 122 (290)
At time 775, value = 126 (294)
At time 785, value = 12a (298)
At time 795, value = 12e (302)
At time 805, value = 132 (306)
At time 1035, value = 128 (296)
At time 1045, value = 11e (286)
At time 1055, value = 114 (276)
At time 1065, value = 10a (266)
At time 1075, value = 100 (256)
At time 1085, value = 0f6 (246)
At time 1095, value = 0ec (236)
At time 1105, value = 0e2 (226)
At time 1115, value = 0d8 (216)
At time 1125, value = 0ce (206)
At time 1135, value = 0c4 (196)
At time 1145, value = 0ba (186)
```

```
At time 1155, value = 0b0 (176)
At time 1165, value = 0a6 (166)
At time 1175, value = 09c (156)
At time 1185, value = 092 (146)
At time 1195, value = 088 (136)
At time 1205, value = 07e (126)
At time 1215, value = 074 (116)
At time 1225, value = 06a (106)
At time 1235, value = 060 (96)
At time 1245, value = 056 (86)
At time 1255, value = 04c (76)
At time 1265, value = 042 (66)
At time 1275, value = 038 (56)
At time 1285, value = 02e (46)
At time 1295, value = 024 (36)
At time 1305, value = 01a (26)
At time 1315, value = 010 (16)
At time 1325, value = 006 (6)
At time 1335, value = 3fc (-4)
At time 1345, value = 3f2 (-14)
At time 1355, value = 3e8 (-24)
At time 1365, value = 3de (-34)
At time 1375, value = 3d4 (-44)
At time 1385, value = 3ca (-54)
At time 1395, value = 3c0 (-64)
At time 1405, value = 3b6 (-74)
At time 1415, value = 3ac (-84)
At time 1425, value = 3a2 (-94)
At time 1435, value = 398 (-104)
At time 1445, value = 38e (-114)
At time 1455, value = 384 (-124)
At time 1465, value = 37a (-134)
At time 1475, value = 370 (-144)
At time 1485, value = 366 (-154)
At time 1495, value = 35c (-164)
At time 1505, value = 352 (-174)
At time 1515, value = 348 (-184)
At time 1525, value = 33e (-194)
At time 1535, value = 334 (-204)
At time 1545, value = 32a (-214)
At time 1555, value = 320 (-224)
At time 1565, value = 316 (-234)
At time 1575, value = 30c (-244)
At time 1585, value = 302 (-254)
At time 1595, value = 2f8 (-264)
At time 1605, value = 2ee (-274)
At time 1615, value = 2e4 (-284)
At time 2035, value = 2e8 (-280)
At time 2045, value = 2ec (-276)
At time 2055, value = 2f0 (-272)
At time 2065, value = 2f4 (-268)
At time 2075, value = 2f8 (-264)
At time 2085, value = 2fc (-260)
At time 2095, value = 300 (-256)
At time 2105, value = 304 (-252)
At time 2115, value = 308 (-248)
At time 2125, value = 30c (-244)
tbcounter-6.v:50: $finish called at 2132 (1ns)
```

Change error behavior

It is highly desirable to notice errors that occur during simulation. Because the default behavior of *Icarus Verilog* is to continue simulation after an assertion fails, we define the macro *myassert* that implements the desired behavior to exit the simulation in Listing 13, Lines 3 to 8. Calls to the *assert* statement are replaced by calls to the *myassert* macro in Lines 69 to 71 and 73.

We choose a *macro* over a *task* in order to preserve expressive error messages with line numbers referring to the failed assertion (a macro is expanded where it is called). When encapsulating *myassert* in a task, an error message would refer to the line number in which *assert* is called from within the task, making it potentially harder to debug the DUT.

Listing 13: Changing simulation behavior in case of an error (*tb_counter-7.v*)

```
1  `timescale 1ns/1ns
2
3  `define myassert(x) "
4     assert(x) "
5     else begin "
6        $error("FAIL: assert(%s) in %s:%0d", `"x`", `FILE, `LINE); "
7        $finishandreturn(1);"
8     end
9
10
11 module tbcounter;
12
13    // Specify dump file and the scope to be dumped
```

```
14    initial begin
15       $dumpfile("counter.vcd");
16       $dumpvars(0, tbcounter);
17    end
18
19    // Define inputs
20    reg clk=0;
21    reg rst=0;
22    reg mode=0;
23
24    // Define outputs
25    wire signed [9:0] cnt;
26
27    // Instantiate design under test (DUT)
28    counter dut (
29       .clk(clk),
30       .rst(rst),
31       .mode(mode),
32       .cnt(cnt)
33    );
34
35    // Generate clock
36    always #5 clk = !clk;
37
38    // Stimulate the design (vary input values)
39    initial begin
40
41       // Reset the DUT
42       #5 rst = 1;
43       #10 rst = 0;
44
45       // Operate the DUT
46       #17 mode = 1;
47       #800 mode = 0;
48       #800 mode = 1;
49       #100 mode = 0;
50
51       $finish;
52
53    end
54
55    initial $monitor("At time %0t, value = %h (%0d)", $time, cnt, cnt);
56
57    // Store past values of signals rst, mode, and cnt
58    reg signed [9:0] pastcnt = 'x;
59    reg pastmode = 'x;
60    reg pastrst = 'x;
61    always @(posedge clk) begin
62       pastcnt ¡= cnt;
63       pastrst ¡= rst;
64       pastmode ¡= mode;
65    end
66
67    // Verify properties of the counter
68    always @(posedge clk) begin
69       `myassert (cnt ¿= -287);
70       `myassert (cnt ¡= 309);
71       `myassert (cnt != 75);
72       if(pastrst) begin
73          `myassert(cnt == 13);
74       end else if (!rst) begin
75          if (pastmode == 1) begin
76
77             // Add the remaining assertions here to check the correct
78             // increment
79
80          end else begin // pastmode == 0
81
82             // Add the remaining assertions here to check the correct
83             // decrement
84
85          end
86       end
87    end
88
89 endmodule
```

Simulating the testbench given in Listing 13, assuming the DUT to implement incorrect reset behavior, results in the output given in Listing 14, which is way more readable than the output given in Listing 12 for the same simulation. Also, the generated debug output given in Lines 5 and 6 is more expressive than in Lines 5 and 6 of Listing 12.

Listing 14: Simulating the testbench given in Listing 11

```
1 $ iverilog -g2012 -l `yosys-config --datdir/simcells.v` -o counter counter.v tbcounter-6.v
2 $ vvp counter
3 VCD info: dumpfile counter.vcd opened for output.
4 At time 0, value = 00e (14)
5 ERROR: tbcounter-7.v:75: FAIL: assert(cnt == 13) in tbcounter-7.v:72
6      Time: 15 Scope: tbcounter
```

Table 4: Functionality of buggy counters

Target	Functionality
0	correctly implemented counter
1	cnt may be greater than maximum
2	cnt may be less than minimum
3	cnt does not jump over invalid
4	increment is 1 larger than specified
5	cnt jumps (up) over invalid to value 1 larger than specified
7	decrement is 1 less than specified
8	cnt jumps (down) over invalid to value 1 smaller than specified
10	initial value is 1 larger than specified

```
7 At time 15, value = 004 (4)
```

Summary

We developed a pretty powerful testbench, ready to be extended and used to verify the conformance of the counter's behavior to its specification.

Your task

Your task is to complete the testbench given in Listing 13 such that it checks the correct behavior of the counter's increment, decrement and invalid value.

The goal of the testbench is that you can use it to verify the implementation of your counter. As you have not yet implemented your counter, we provide you with a sample implementation *counter.v*, which you can use to develop your testbench.

In order to enable you debugging your assertions, we provide you with a set of test counters, almost all of them violating properties specified for our counter. When you simulate your testbench along with each of the test counters, an assertion should fail and the simulation should exit (except for *counter-0.v*). This means that your assertions cover that error, which is good for your testbench.

Along with the set of test counters, we provide a *Makefile* to increase usability, and to speed up the verification process. The targets defined in the Makefile are numbers 0, 1, 2, 3, 4, 5, 7, 8, 10. Table 4 summarizes the buggy counters' functionality.

It is fairly straight-forward to execute the simulations of buggy counters using your testbench. In the command-line terminal, change to the directory of your Makefile, and run the command given in Listing 15.

Listing 15: Simulate test counter 10

```
1 make 10
```

Submission details

To submit your testbench, please attach your completed version of *tb_counter.v* to your submission e-mail.